**RESEARCH QUESTIONS**

\*RTSO = Regression Test Suite Optimization

|  |  |  |  |
| --- | --- | --- | --- |
| **RQs** | **RQ Statement** | **Motivation** | **Metrics** |
| **RQ 1** | ***What is the status of the research in RTSO approaches?*** | | |
| **RQ 1.1** | *What are the research trends in RTSO?* | Identify whether **interest** in the field is growing, stagnating, or declining, revealing **shifts** in methodological paradigms. | 1. Pub Year 2. Method: {Selection, Minimization, Prioritization} |
| **RQ 1.2** | *What are the similarities and differences between RTSO approaches?* | **Comparing** approaches through their goals, assumptions, and trade-offs, making **informed decisions** on which method to use based on the problem. | Coding starting from the taxonomy by Yoo et al., optionally extended if needed |
| **RQ 1.3** | *Which type of algorithms are used to solve RTSO problems?* | Shedding light on the **algorithm families** employed to solve RTSO problems. | {Search-based algorithms, Quantum-search-based algorithms,  ML-based algorithms} |
| **RQ 1.3a** | *When applying search-based algorithms, how many objectives are considered?* | Shedding light on how well current techniques capture the **complexity** of real-world **constraints** and **objectives**. | {Single-, Two-, Three-, Multi-} |
| **RQ 2** | ***What types of SUT are used in RTSO experiments?*** | | |
| **RQ 2.1** | *What datasets/SUTs are the most used in RTSO techniques?* | Promoting **reproducibility** and highlighting potential **over-reliance** on specific datasets, which can threaten external validity. | Dataset Name |
| **RQ 2.2** | *What are the main characteristics of the commonly used SUT?* | This question clarifies whether research targets **specific** **domains** or is **broadly applicable**, if proposed methods and tools solve **real-world scenarios,** and so on. | 1. Application Domain 2. # of Test Cases 3. LOC |
| **RQ 2.3** | *Which features of the previously listed datasets are leveraged by RTSO techniques?* | The criteria determine the optimization performance. Understanding the used features allows for a better **interpretation** of algorithms and **fair comparisons**. | {[statement/branch/…/ method] coverage, execution time, fault detection history, failure rate,...} + |
| **RQ 3** | ***How have empirical evaluations of RTSO techniques been conducted?*** | | |
| **RQ 3.1** | *What metrics are used to evaluate RTSO techniques?* | Investigating what the community values and helping detect **gaps** in **assessment** practices. | {Hypervolume, APFD, …} |
| **RQ 3.2** | *How is reproducibility ensured in RTSO research?* | Assessment of **reproducibility** in RTSO research. | 1.0: Full replication package (scripts+datasets+instructions)  0.5: Partial artifact or insufficient instructions  0.0: No code, no artifact available |

**TEST CASE PRIORITIZATION TAXONOMY**

Immagine che contiene testo, schermata, diagramma, Parallelo

Il contenuto generato dall'IA potrebbe non essere corretto.

**Coverage-Based:** ranks test cases based on how much structural code coverage they provide. Variants include statement and branch coverage, and Fault Exposing Potential (FEP). These techniques are typically evaluated using the APFD metric, which quantifies how quickly faults are detected.

**Model-Based:** prioritizes/selects test cases using behavioral models of the system under test. When a model is updated due to software changes, test cases are classified into high-priority (TSH) if they are affected by the modification, and low-priority (TSL) otherwise. The initial strategy prioritizes TSH randomly, followed by TSL. More advanced versions of the approach incorporate dependency analysis, allowing prioritization to consider both direct and indirect impacts of changes within the model.

**Requirement-Based:** prioritizes/selects test cases according to the software requirements they cover. Each test case is linked to one or more requirements, and optimization is based on attributes of those requirements, such as customer-assigned priority or implementation complexity.

**Probability-Based:** uses statistical and probabilistic models to guide the ordering/selection of test cases. This approach can leverage execution history, selecting tests that have not been run recently (e.g., using Least Recently Used heuristics); it can use Bayesian Networks to estimate fault detection likelihood based on code changes and test coverage, dynamically updating probabilities during execution; some applies matrix decomposition (SVD) to identify clusters of frequently co-modified files, aligning them with test cases and recent code modifications.

**Distribution-Based:** Distribution-based techniques minimize and prioritize test cases based on the distribution of the profiles of test cases in the multi-dimensional profile space. Test case profiles are produced by the dissimilarity metric, a function that produces a real number representing the degree of dissimilarity between two input profiles. Using this metric, test cases can be clustered according to their similarities.

**Human-Based (Clustering):** integrates human intuition with machine learning. Testers provide pairwise comparisons between test cases, indicating which should be executed first. These preferences are used by a learning-to-rank algorithm to generate a prioritization/selection function. Initial heuristics like statement coverage and cyclomatic complexity support the learning process.

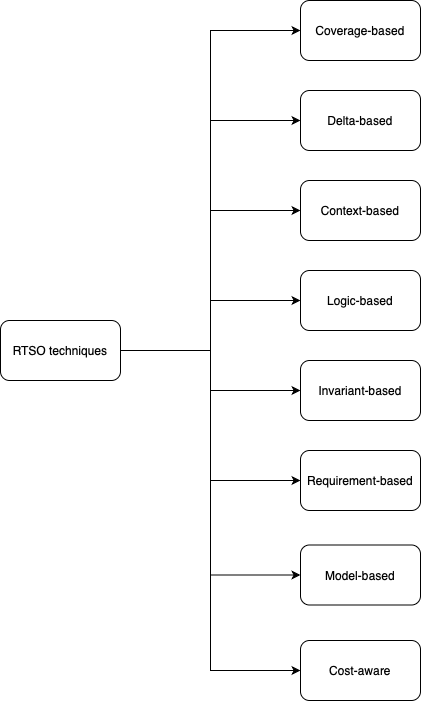
**Human-Based (Non-Clustering):** To improve scalability, clustering is introduced in the Human-based approach: testers rank clusters instead of individual tests (inter-cluster), while tests within each cluster are ordered automatically (intra-cluster).

**History-Based:** prioritizes/selects test cases based on historical change patterns among software artefacts. Using Singular Value Decomposition (SVD) on a change matrix, it identifies association clusters—groups of files frequently modified together. Each file is linked to test cases that affect or execute it. When a new system change occurs, it is encoded as a modification vector, and test cases are prioritized/selected based on how closely they relate to the changed files via their cluster associations.

**Cost-Aware:** addresses resource constraints by considering test execution costs and fault severities. Cost-aware approaches optimize for limited budgets. Metrics like APFDc incorporate both cost and severity. Some techniques (e.g., time-aware prioritization) select a test subset that fits within a time limit using search-based methods like genetic algorithms. Others apply multi-objective optimization (e.g., Pareto-front) or ILP to balance trade-offs. These methods aim to detect severe faults early and efficiently, and are especially valuable when only partial testing is feasible due to time or resource limits.

**Others:** includes a variety of specialized techniques. Mutation-based targets interface contracts in component-based systems. Session-based uses real user sessions to test web applications, leveraging criteria like HTTP request count or 2-way parameter-value coverage. Property-based approaches prioritize/select tests that could violate model properties, as determined by model checkers. Data-flow and call-tree models rank tests by def-use pairs or call-path coverage, respectively. Slice-based methods prioritize/select tests with large or relevant program slices.

**TEST CASE MINIMIZATION TAXONOMY**



\*See Model-based, Requirement-based and Cost-aware approaches’ descriptions from previous Taxonomy.

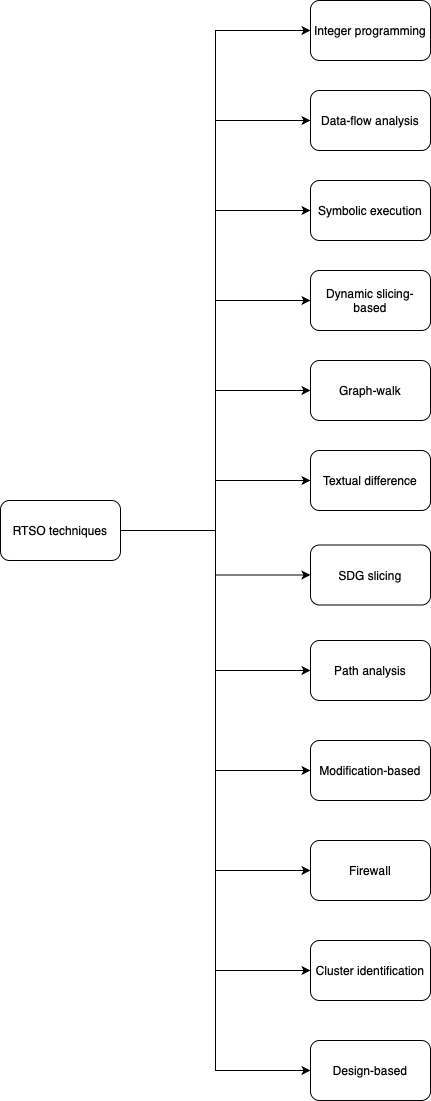
**Delta-based:** The delta-based approach aims to simplify a single failing test case by removing unnecessary steps or inputs while preserving its fault-revealing behavior. This is particularly useful for debugging complex, auto-generated test cases. Techniques like delta debugging iteratively reduce test case size without losing the failure, enabling easier fault localization. Unlike prioritization, this approach focuses on individual test case simplification, not suite-wide ranking.

**Context-based:** The context-based approach assumes that part of the system under test (the context) is fault-free. When the system is modeled as a composition of two FSMs (component and context), test cases covering only the context can be safely removed. This enables safe minimization by eliminating test cases that cannot reveal faults in the component under test. This assumption-based strategy is unique to minimization and does not apply to prioritization.

**Logic-based:** The logic-based approach reduces test suites by exploiting formal hierarchies of Boolean fault classes. If detecting one class guarantees detection of another (via logical subsumption), only the stronger class needs to be tested. Test cases are selected accordingly, ensuring that fault detection is preserved while minimizing redundancy. This method provides formal guarantees but is applicable mainly to predicate logic and Boolean testing scenarios.

**Invariant-based:** The invariant-based approach minimizes test suites by preserving dynamically inferred program invariants. Using tools like Daikon, it detects semantic properties during test execution. A test case is removed only if its absence does not alter these invariants. This ensures that the reduced suite maintains behavioral validity, often with better fault-detection capability than coverage-based minimization. It prioritizes semantic preservation over structural reduction.

**TEST CASE SELECTION TAXONOMY**

****

**Integer Programming:** The integer programming-based approach selects the minimal-cost subset of test cases that covers all program segments affected by a change. Each test is a binary variable in an optimization model constrained by segment coverage and reachability. The model ensures that every impacted segment is exercised by at least one selected test. While it offers optimality guarantees, it is limited by its sensitivity to control-flow changes, which can require reanalysis of all tests.

**Data-flow analysis-based:** The data-flow analysis-based approach selects test cases that cover changed or affected definition–use pairs in the updated program. These pairs model how data flows through the code, helping to identify where behavior might change. Enhancements include incremental analysis and slicing to reduce cost. While precise for data-dependent logic, the approach may miss control-related or output-only changes. It has also been adapted to domains like spreadsheets through interactive visual frameworks.

**Symbolic execution:** The symbolic execution-based approach selects test cases that traverse control-flow paths reaching modified code. It symbolically executes test inputs over a control-flow graph, discarding tests that cannot reach changes. Input partitioning ensures path coverage across input domains. This method is highly precise but limited by the computational cost of symbolic execution and difficulties in handling complex constructs like pointer arithmetic or recursion.

**Dynamic-slicing:** The dynamic slicing-based approach selects test cases based on which executed statements influence program outputs. It uses dynamic slices—subsets of execution traces—to identify if a test case is affected by code changes. Enhancements include relevant and approximate relevant slices, which consider conditional branches and indirect effects. This approach improves precision over static or execution slicing, but may miss changes unrelated to data or control dependencies. It is most effective when the program's control-flow structure remains stable.

**Graph-walk:** The graph-walk-based approach selects regression test cases by traversing graph representations (e.g., CDG, PDG, CFG, ICFG) to identify control/data flow changes. Test cases are retained if they execute affected nodes or edges. The method supports interprocedural, object-oriented, and component-based systems, and can incorporate behavioral models like activity diagrams or state charts. It is widely adaptable and supports black-box scenarios, though its precision depends on graph detail and accurate change mapping.

**Textual-difference:** The textual difference-based approach selects regression tests by comparing source code text between versions using tools like “diff”. After normalizing the code to eliminate cosmetic changes, the modified lines are mapped to test cases that execute them. Although simple and language-agnostic, this method behaves similarly to CFG-based graph-walk approaches but lacks semantic depth. It is efficient but may over-approximate test impact due to its purely textual nature.

**SDG slicing:** The SDG slicing-based approach selects test cases by comparing slices from System Dependence Graphs of two program versions. Statements are classified based on common execution patterns, and test cases are reused if they cover statements with equivalent behavior across versions. Affected statements trigger the selection of related tests. This interprocedural method captures deep structural changes, but is not safe—it cannot detect impacts from deleted code, which may affect program behavior.

**Path analysis:** The path analysis-based approach selects test cases based on comparisons of algebraically represented execution paths between program versions. Paths are classified as modified, new, cancelled, or unmodified. Only test cases covering modified paths are selected, ignoring new or cancelled ones. This may omit fault-revealing tests, making the approach unsafe. Though effective for structural path comparison, its precision depends on how "modification" is defined.

**Modification-based:** The modification-based approach selects test cases by tracking which program entities (functions, variables, macros) have changed between versions. Tools like TestTube monitor test executions to establish mappings between test cases and these entities. Tests that interact with modified entities are re-executed. The method is safe, but its accuracy may be compromised in languages with pointer arithmetic or implicit type operations, where changes may be hard to detect reliably.

**Firewall-based:** The firewall-based approach selects test cases by drawing a boundary around modified modules and their interacting components. Modules are categorized based on whether code or specifications have changed. Test cases that execute within or across this "firewall" are retained to ensure integration correctness. The method is safe but conservative, potentially over-selecting tests. It has been applied to OO systems, GUIs, and black-box components, especially in integration testing scenarios.

**Cluster identification:** The cluster identification-based approach partitions the program’s CFG into single-entry, single-exit clusters and identifies modifications by comparing clusters across versions. Modified clusters are marked as MOD, and test cases that traverse these are selected. This ensures safe regression test selection, regardless of the change type. However, clusters may over-approximate the modification scope, reducing precision by including unrelated test cases.

**Design-based:** The design-based approach selects regression test cases by analyzing changes in UML design models. Using automated impact analysis and assuming traceability to test cases, it categorizes them as obsolete, retestable, or reusable. It enables early, black-box RTS and scales well for large systems. Its precision and usefulness depend on the existence of reliable traceability links between design elements and test cases.